Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2017

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Integrante	LU	Correo electrónico
Nicolás Ippolito	724/14	ns_ippolito@hotmail.com
Emiliano Galimberti	109/15	emigali@hotmail.com
Gregorio Freidin	433/15	gregorio.freidin@gmail.com
Pedro Mattenet Riva	428/15	peter_matt@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Utilidades en la vida Real

1.1 El Grupo Social Más Famoso

Modelando a las personas como nodos, si una persona comparte un grupo con otra, sus respectivos nodos están conectados por una arista. Si la persona está conectada a muchas otras, o bien que el grado de su nodo es alto, decimos que es $famosa$. Se busca encontrar el grupo social más famoso de todos, esto es, el grupo cuyos integrantes están conectados a más personas fuera de éste. Notar que un grupo puede estar formado por sólo una o dos personas

1.2 El Mercado Más Influyente

Representando comercios como nodos, si un comercio tiene negocios con otro, entonces sus nodos estarán unidos por una arista (no importa quién vende o quién compra, las aristas no tienen dirección). Se denomina *mercado* al subconjunto de comercios entre los cuales todos hacen negocios entre sí (sería nuestra clique). Un comercio es $influyente$ si tiene muchos negocios con otros comercios. Se busca entonces, el Mercado Más Influyente. Esto es, el subconjunto de comercios (que son mercado) que más negocios poseen con los comercios de afuera del mercado.

1.3 El Grupo Más Toxico

En un conocido juego MOBA online, la compañía creadora del juego (RITO GAMES) nos pidió ayuda. Se dice que un jugador es **tóxico** cuando insulta, agrede o discrimina a otro jugador. Generalmente, como los jugadores son de poca edad madurativa, al ser *entoxicados* por un jugador devuelven los insultos, generando un enlace entre los dos nodos, de manera tal que el jugador del nodo A y el jugador del nodo B están unidos por una arista sin dirección cuando ambos son tóxicos entre si. Se desea encontrar al subconjunto de jugadores, todos tóxicos entre si, que insulten, agredan o discriminen a más jugadores de afuera del grupo.

1.4 BANE ATACA DE NUEVO!

Así como dice el título, Bane ataca de nuevo! El joven villano de la saga DC nacido en la República Caribeña de Santa Prisca se quedó frustrado por su intento fallido de destruir Gotham City ya atrás en el 2012.

Con una perfecta paciencia estuvo esperando el momento indicado para anunciar sus intenciones de regreso, y tras la muerte del histórico Batman (W.W.Anderson), más bien conocido como Adam West, sabe bien que ya nadie podrá detenerlo.

Sin embargo esta vuelta no logró conseguir la Visa para volver a los Estados Unidos. Por lo que sin mucha sorpresa, al ser conocida la historia de enemistad de este personaje con las pastas, su siguiente objetivo para aterrorizar ha pasado a ser la bella Italia.*****

Así bien la noticia en particular llegó a la vieja Venecia, cuyas autoridades sin más preámbulo, como política de seguridad, decidieron reacomodar su demografía de forma que si Bane llega a atacarla, le cueste mucho cubrir todos los caminos de salida.

Prontamente salieron condiciones de cómo se debería llevar esto a cabo esto, principalmente una. Que todas las islas a las que se muden estén interconectadas entre sí. Se quiere que la economía funcione mas allá de todo mal. Y les pareció que esta idea de interconexion entre ella sería la más simple para lograrlo.

Rápidamente un joven despierto les aviso que este problema no tiene una solución exacta que se calcule de forma polinomial. Por lo que se abrió la búsqueda de quien les pueda aportar la mejor heurística para resolver su problema de reacomodamiento demográfico. Y la recompensa será MIL pizzas!

1.4.1 Modificación BANE

Realizamos una modificación a *Bane ataca de nuevo!* modificado desde el *****

Bane puede atacar todas las ciudades de Italia que quiera, pero sabe que debe intentar controlar su maldad pues mientras más ciudades ataque, más probabilidades tiene de fracasar en su cometido. Por otro lado, tiene una extraña obsesión con derribar puentes. Afirma que es porque le gusta que la gente víctima de su ataque quede encerrada (pero existe el rumor de que es una especie de fetiche sexual).

Luego, lo que busca Bane es atacar un conjunto de ciudades que estén conectadas entre sí, tales que la suma de todos sus puentes que las conectan con otras ciudades que no están en ese conjunto sea máxima. De esta forma, derribará muchísimos puentes y la gente se quedará encerrada entre ciudades que estarán controladas por él.
Si consigue esto, podrá llegar al pico de su maldad y logrará la iluminación divina.

Abstrayéndonos de lo descripto, lo que se busca es encontrar la clique de frontera máxima (siendo nodos las ciudades y aristas los puentes).

2 Algoritmo Exacto

Para resolver el problema descripto de forma exacta se resolvió utilizar la técnica de backtracking.

Teniendo los nodos numerados del 1 al n, partimos desde el nodo *1* ($i = 1$) y en cada paso:

- Utilizamos el nodo i
- No utilizamos el nodo i

De esta forma se arma un árbol de recursión que en cada hoja contiene una combinación específica de nodos utilizados y nodos no utilizados. Notar que si no hubiesen podas, las hojas del último nivel serían todas las combinaciones posibles (2^n).

Al llegar a cada hoja, analizaremos si esa hoja es una cliqué y calcularemos su frontera, quedándonos con la que sea menor entre todas las posibles combinaciones de nodos que son cliqués.

A su vez, se buscó reducir un poco el tiempo de cómputo para facilitar las experimentaciones, por lo que se implementó una poda que consiste en guardar el último nodo (*ultNod*) utilizado de cada rama y luego, si al estar parado en el nodo i el mismo no es adyacente a *ultNod*, entonces me olvido de la rama que correspondería a utilizar al nodo i ya que ésta no va a ser una cliqué.
También para podar aún más se podría no sólo mirar si el nodo i es adyacente a *ultNod*, sino también chequear que sea adyacente a todos los utilizados en esa rama. Se eligió la primera de las descriptas dado que ésta se puede chequear en tiempo constante (lo que no implica que vaya ser más eficiente, pero dado que las heurísticas son mucho más demandantes e interesantes a la hora de experimentar, se decidió no indagar tanto buscando la mejor poda para el algoritmo exacto).

2.1 Pseudocódigo y complejidad teórica

La idea del *Algoritmo 1* es crear todo lo necesario para que el algoritmo recursivo (*CliqueMaxFrontAux*) resuelva el problema.

Se decidió utilizar una matriz de adyacencia para representar al grafo.

Algorithm 1 CliqueMaxFront

```
procedure CliqueMaxFront(Graph grafo)
    n ← cantidad de nodos del grafo
    vecRes ← vector de longitud n
    vecAux ← vector de longitud n
    res ← 0
    i ← 0
    ultUsado ← −1
    CliqueMaxFrontAux(grafo, n, i, ultUsado, vecAux, res, vecRes)
    tamClique ← cantUnosVec(vecRes)
    return (res, tamClique)
```

Los dos vectores creados (*vecRes* y *vecAux*) serán vectores tales que v$[i] = 1$ sii el nodo i es utilizado en esa rama recursiva; *vecRes* contendrá finalmente los nodos que pertecenen a la cliqué máxima y *vecAux* será un vector auxiliar que servirá para guardar los nodos utilizados en cada rama. Recordar que estamos recorriendo todas las combinaciones posibles de nodos y definiendo su frontera (de ser cliqué esa combinación).

Luego de la función recursiva que hace los cálculos, para definir la cantidad de nodos de la cliqué de máxima frontera utilizamos la función *cantUnosVec* que cuenta la cantidad de unos de *vecRes* de forma lineal.
Analizando el *Algoritmo 2* vemos que en el peor caso -grafo completo, donde todo par de nodos es adyacente- en cada paso hago dos llamadas recursivas (usar y no usar el nodo) por lo que en total haría 2^n llamadas recursivas, cada

una de ellas de complejidad constante. Al llegar a una hoja realizo dos operaciones cuadráticas (chequear si es cliqué y calcular su frontera). Luego, al llegar a cada una de las 2^n hojas voy a tener que realizar una o dos operaciones de complejidad cuadrática.

Finalmente, la complejidad es $O(2^n * n^2 + n) = O(2^n * n^2)$.

Algorithm 2 CliqueMaxFrontAux

```
procedure CliqueMaxFrontAux(Graph grafo, int n, int ite, int ultUsado, vector<int> vecAux, int res, vector<int> vecRes)
    if ite = n then
        if esClique(grafo, vecAux) then
            cant ← CalculoFrontera(grafo, vecAux)
            if cant > res then                          ▷ Si la frontera de esta rama era mejor, la actualizo
                res = cant
                vecRes = vecAux
    else
        if Los nodos ite y ultUsado no son adyacentes then
            CliqueMaxFrontAux(grafo, n, ite + 1, ultUsado, vecAux, res, vecRes)     ▷ Solo rama de no usar nodo
        else
            CliqueMaxFrontAux(grafo, n, ite + 1, ultUsado, vecAux, res, vecRes)     ▷ Rama de no usar nodo
            vecAux[ite] = 1
            CliqueMaxFrontAux(grafo, n, ite + 1, ite, vecAux, res, vecRes)          ▷ Rama de usar nodo
```

Algorithm 3 EsClique

```
procedure EsClique(Graph grafo, vector<int> vector)
    if cantUnosVec(vector) = 0 then
        return false
    for i = 0 to longitud de vector do
        if vector[i] = 1 then
            for j = i + 1 to longitud de vector do
                if vector[j] = 1 and i y j no son adyacentes en grafo then
                    return false
    return true
```

Algorithm 4 calcFrontera

```
procedure calcFrontera(Graph grafo, vector<int> vector)
    res ← 0
    cantNodClique ← cantUnosVec(vector)
    for i = 0 to longitud de vector do
        if vector[i] = 1 then
            res = res + gradoNodo(grafo, i) − cantNodClique + 1
    return res
```

Donde *gradoNodo(grafo, i)* es una función de complejidad lineal que recorre la fila (de la matriz de adyacencia del grafo) que corresponde al nodo i y ve cuántos nodos adyacentes tiene.

La idea del cálculo de la frontera es ver para cada nodo cuántos adyacentes tiene que no sean de la cliqué y sumarlos.

2.2 Experimentación

Los experimentos fueron realizados en una computadora con un procesador I5 4440 y 8GB de memoria RAM.

Sea V la cantidad de nodos, en la experimentación se generaron los siguientes casos o familias de grafos:

- **T1**: Grafos que tienen V aristas.
- **T2**: Tienen al rededor de $V * \frac{V-1}{4}$ aristas. En otras palabras, tienen aproximadamente la mitad de las aristas que tendría un grafo completo.
- **T3**: Son grafos completos. Luego, en la matriz de adyacencia de esta familia de grafos G[i][j] = 0 sii i = j.

En las familias *T1* y *T2*, para determinar las adyacencias entre nodos se utilizó la librería random de C++ (siempre corrigiendo el caso en el cual estás buscando una adyacencia para el nodo i y random te devuelve i). En consecuencia, se aleatorizan las posibles cliqués de cada uno de éstos y, en particular, se hace difícil dar un número exacto de aristas en el caso *T2* -por lo que para describirlo se especificó que tienen "al rededor" de cierta cantidad de aristas-.

Para cada caso se crearon 40 instancias con distinta cantidad de nodos, desde 2 hasta 24 (se frenó en ese número ya que los grafos completos de 25 nodos tenían un tiempo de cómputo muy grande).

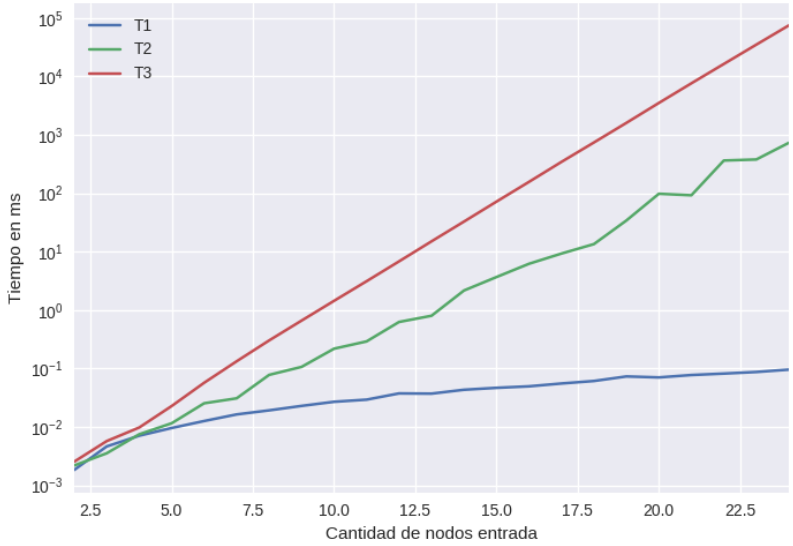


Figure 1: Tiempos con respecto a la cantidad de nodos de la entrada - escala logarítmica

Analizando la **Figura 1** vemos varias cosas:

- *Los tiempos de cómputo del caso T3 son mucho más grandes que el resto.* Esto es completamente lógico y se debe a que para esta familia de grafos la poda no sirve ya que los mismos son completos. Luego, en cada rama recursiva se va a llegar al fondo del árbol (excepto en una: la combinación que corresponde a no utilizar ningún nodo) por lo que habrá $2^n - 1$ hojas en el último nivel donde cada una tendrá que pagar el costo n^2 de ver si es una cliqué y de calcular su frontera.
- *Los tiempos de cómputo del caso T1 son muy pequeños en relación a los otros.* Tiene sentido ya que al haber pocas aristas la poda sería muy eficiente.
- *En el caso T2 el crecimiento parece presentar algunas irregularidades.* Esto es producto de la forma en la que fueron generadas las instancias, ya que al determinarse las adyacencias de forma aleatoria (y también dado que el número exacto de aristas varía para instancias con la misma cantidad de nodos) tanto la eficiencia de la poda como la cantidad de cliqués varía mucho, causando una cantidad muy variable de pasos recursivos y de hojas que sean cliqués (y por ende sumando al tiempo de cómputo la operación cuadrática para calcular su frontera) según cada instancia. En particular se ve que pareciera no haber variaciones de tiempos de cómputo en las cantidades de nodos 20 a 21, así como 22 a 23, por lo cuál se añade la **Figuras 2** para demostrar que no es así.

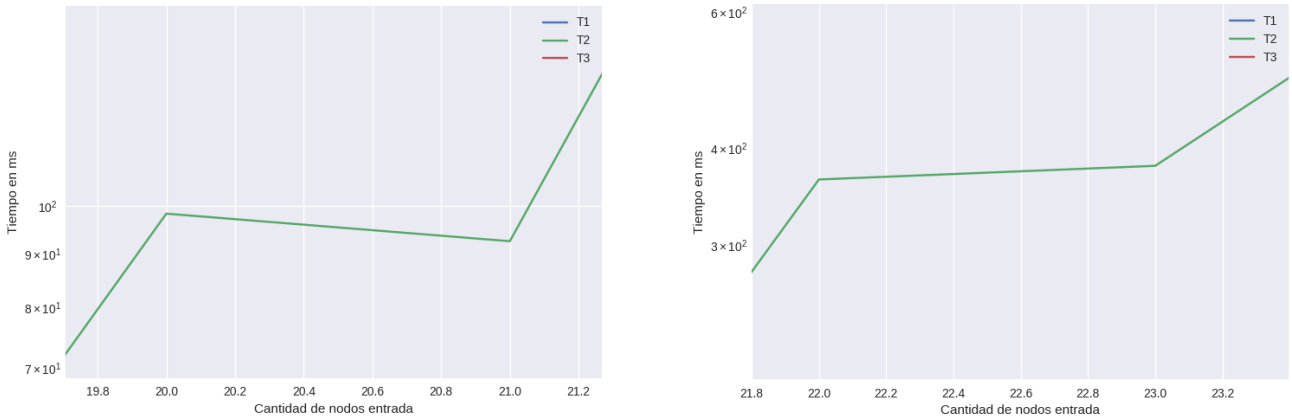


Figure 2: Tiempos con zoom en cantidad de nodos 20 a 21 y 22 a 23

3 Heurística Golosa

En esta sección se plantean dos heurísticas golosas que tienen menor complejidad temporal que el algoritmo exacto, pero que no obtienen necesariamente un resultado óptimo.

3.1 MFCGM

La heurística *MFCGM* -máxima frontera cercana al grado máximo- consiste en lo siguiente:

- **Paso 1**: Buscar el nodo de grado máximo *nodoGM* (o uno de ellos si fuesen varios). Notar que ésta ya es una posible clique de máxima frontera.
- **Paso 2**: Recorrer todos los nodos adyacentes a *nodoGM* y calcular la frontera que forma el conjunto de nodos [nodoAdy, nodoGM] siendo *nodoAdy* adyacente a *nodoGM*. Luego, habrá un conjunto de 2 nodos que va a ser el de mayor frontera de todos los analizados.
  - Por un lado ese conjunto, al que llamo *2nodoFrontMax*, me lo voy a guardar ya que lo voy a necesitar para el siguiente paso.
  - Por otro lado, si *2nodoFrontMax* tenía mejor frontera que mi solución actual (la frontera de *nodoGM*), actualizo la mejor solución.
- **Paso 3**: Busco extender *2nodoFrontMax*. La idea es agregarle nodos uno por uno, siempre añadiendo aquél que en la iteración i forma una clique de i nodos (o sea, es adyacente a todos los que ya añadiste anteriormente) y que además forma el conjunto de máxima frontera con i nodos. Para esto no visitamos los vecinos de todos los nodos de la clique actual, sino sólo los vecinos del nodo agregado en la iteración $i-1$.
  Luego, en realidad lo que estamos encontrando es la clique de máxima frontera de i nodos tal que el nodo agregado en la iteración i es adyacente al nodo agregado en la iteración $i-1$, y no la clique de máxima frontera de i nodos de todo el grafo.
  Este proceso continúa hasta llegar a una iteración j tal que el nodo agregado en la iteración $j-1$ no tiene vecinos que formen una clique con aquellos ya agregados anteriormente.
  A su vez, siempre que encuentre la máxima frontera de k nodos de la que hablé arriba, si ésta es mayor que la solución actual, actualizo la misma.

Al finalizar estos 3 pasos tengo una solución que se generó de forma golosa con un algoritmo de complejidad polinomial (se analizará más adelante).
No necesariamente será óptima, lo que también quedará claro en la sección **3.1.1**.

3.1.1 Instancias no óptimas

Vamos a presentar un tipo de grafos en el cual la heurística *MFCGM* no devuelve la solución óptima.

La familia de grafos que analizaremos tiene las siguientes características:

- Tiene dos componentes conexas.
- Una de las dos componentes conexas es una estrella (un nodo de grado de grado d adyacente a d nodos de grado 1).
- La otra componente conexa tiene dos nodos v y u adyacentes entre sí, donde además cada uno de ellos tiene otros $d-2$ nodos adyacentes de grado 1.

A este tipo de grafos los llamaremos **Estrella no óptima**. Abajo ilustramos un ejemplo de una instancia con 14 nodos.

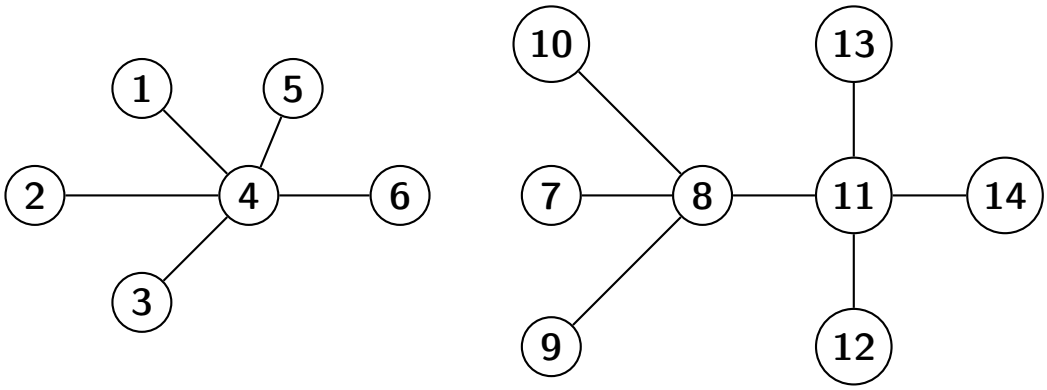


Figure 3: Instancia de 'Estrella no óptima' con 14 nodos

Analizando la **Figura 3** se ve que nuestra heurística se quedaría con la componente conexa de la izquierda (pues aquí está el nodo con mayor grado) y finalmente el algoritmo devolvería como resultado la clique de frontera máxima que está en esa componente -el conjunto de nodos [4] cuya frontera es 5-. Por otro lado, vemos que la clique de frontera máxima en realidad está en la componente de la derecha y es el conjunto de nodos [8, 11] que tiene frontera 6. Luego, la heurística NO devuelve el resultado óptimo.

A su vez, vemos que partiendo de $n = 14$, a medida que aumentamos la cantidad de nodos del grafo la heurística devuelve cada vez soluciones más alejadas de la óptima. Sean *v, u* los nodos 8 y 11 respectivamente, y sea x el nodo 4:

- Si agregamos un nodo, lo ponemos como adyacente de cualquier nodo *v'* adyacente a v (distinto de u).
- Si agregamos dos nodos, sean *v'* y *u'* dos nodos adyacentes a v y u respectivamente (y ambos distintos de v y u), uno de los dos nodos agregados será adyacente a *v'* y el otro será adyacente a *u'*.
- Si agregamos tres nodos, uno de ellos será adyacente a x, otro adyacente a v y el restante adyacente a u. De esta forma, ahora la frontera de la clique [x] aumenta en 1, mientras que la frontera de la clique [v, u] aumenta en 2, por lo que la solución devuelta por la heurística se aleja aún más de la óptima.

Así, partiendo de una instancia que tenga una cantidad de nodos n tal que n $\equiv 2 \pmod 3$ y $n \geq 14$, se puede ir construyendo instancias más y más grandes (con respecto a la cantidad de nodos) que vayan siendo iguales en cuanto a la lejanía del resultado óptimo (si añado 1 o 2 nodos) o peores si añado más.

Por último observamos que si la cantidad de nodos es menor a 14, la familia de grafos *Estrella no óptima* no garantiza que la solución de nuestra heurística NO sea óptima, ya que con menos nodos no existe el caso en el cual el nodo x de la componente conexa estrella sea el único nodo de grado máximo y a la vez no sea la clique de frontera máxima (o una de ellas).

3.1.2 PseudoCódigo y Complejidad teórica

Decidimos representar al grafo como dos enteros (n y m que representan la cantidad de nodos y de aristas respectivamente) y una matriz (la matriz de adyacencia del grafo).

A continuación presentaremos el pseudocódigo del algoritmo y justificaremos su complejidad teórica. Aclaramos que los vectores utilizados son como los del algoritmo exacto (representan un sobconjunto de nodos).

Algorithm 5 MFCGM

```
1: procedure MFCGM(Graph grafo)
2:     res ← 0
3:     vecAux ← vector de longitud n inicializado en 0
4:     vecRes ← vector de longitud n inicializado en 0
5:     MFCGMAux(grafo, res, vecAux, vecRes)                ▷ O(n^4)
6:     return (res, vecRes)
```

Algorithm 6 MFCGMAux

```
1:  procedure MFCGMAux(Graph grafo, int res, vector<int> vecAux, vector<int> vecRes)
2:      nodoGM ← nodoGradoMaximo(grafo)                                   ▷ O(n^2)
3:      vecAux[nodoGM] = 1                                                ▷ O(1)
4:      mejorFront ← calcFrontera(grafo, vecAux)                          ▷ O(n^2)
5:      vecRes = vecAux                                                   ▷ O(n)
6:      vecSolIte ← vector de longitud n inicializado en 0                ▷ O(n)
7:      posFrontMejor ← 0                                                 ▷ O(1)
8:      ultNodoAg ← nodoGM                                                ▷ O(1)
9:
10:     agregarVecinoMejorFrontera(grafo, vecAux, vecSolIte, posFrontMejor, nodoGM, ultNodoAg)   ▷ O(n^3)
11:     if posFrontMejor > mejorFront then                                ▷ O(n)
12:         vesRes = vecSolIte                                                                    ▷ O(n)
13:         mejorFront = posFrontMejor
14:     vecAux = vecSolIte                                                ▷ O(n)
15:     ultNodoAux ← ultNodoAg                                            ▷ O(1)
16:
17:     while Hay algún vecino de ultNodoAg que forma clique con todos los anteriores do   ▷ O(n^4)
18:         posFrontMejor = −1                                            ▷ O(1)
19:         for int i = 0 to grafo→n do                                                           ▷ O(n^3)
20:             if sonAdyacentes(grafo, i, ultNodoAg) ∧ noEstabaEnClique(grafo, vecAux, i) ∧
21:               nodoFormaClique(grafo, vecAux , i) then                                         ▷ O(n))
22:                 vecAux[i] = 1                                         ▷ O(1)
23:                 frontAux ← calcFrontera(grafo, vecAux)                ▷ O(n^2)
24:                 if frontAux > posFrontMejor then                      ▷ O(n)
25:                     posFrontMejor = frontAux                                                  ▷ O(1)
26:                     vecSolIte = vecAux                                                        ▷ O(n)
27:                     ultNodoAux = i                                                            ▷ O(1)
28:                 vecAux[i] = 0                                         ▷ O(1)
29:         if posFrontMejor > mejorFront then                            ▷ O(n)
30:             vecRes = vecSolIte                                                                ▷ O(n)
31:             mejorFront = posFrontMejor                                                        ▷ O(1)
32:         vecAux = vecSolIte                                            ▷ O(n)
33:         ultNodoAg = ultNodoAux                                        ▷ O(1)
34:
35:     res = mejorFront                                                  ▷ O(1)
```

En el **Algoritmo 6** se pueden divisar los tres pasos que describimos en la **sección 3.1**, donde el *Paso 1* consiste en la parte del código antes de la función *agregarVecinoMejorFrontera* -lineas 2 a 8-, el *Paso 2* comienza con esa función y termina antes del while -lineas 10 a 15- y el *Paso 3* comprende todo lo que es el while hasta el final -lineas 17 a 35-. Analizaremos la complejidad de cada uno de estos pasos por separado (con alguna explicación para aclarar más el pseudocódigo en algunos casos). Finalmente, la complejidad teórica final será $O(complej(Paso1) + complej(Paso2) + complej(Paso3))$

Paso 1:

Primero explicamos las dos funciones que utilizamos: *nodoGradoMaximo* que encuentra al nodo de grado máximo en el grafo (o uno de ellos si hay más de uno) y lo hace de forma cuadrática ya que recorre linealmente todos los nodos y para cada uno de ellos calcula su grado (que es lineal pues hay que recorrer la fila correspondiente a ese nodo en la matriz de adyacencia); y *calcFrontera* que es la misma que utilizamos para el algoritmo exacto (*Algoritmo 4*) y es cuadrática. Notar que la frontera en este caso es igual al grado de *nodoGM*, pero se decidió utilizar la función *calcFrontera* pues a la hora de ver el pseudocódigo queda más explícito lo que se busca exáctamente (y no afecta a la complejidad).

La complejidad del *Paso 1* es $O(n^2)$ pues las otras operaciones son menos costosas (crear/copiar vectores y enteros).

Paso 2:

La función *agregarVecinoMejorFrontera*, que tiene complejidad $O(n^3)$ y cuyo pseudocódigo está especificado más abajo, modifica a:

- *vecSolIte*: Ahora contendrá al subconjunto de 2 nodos con mejor frontera.
- *posFrontMejor*: Será la frontera de *vecSolIte*
- *ultnodoAg*: Ahora corresponde al nodo vecino de *nodoGM* que formó la mejor frontera de 2 nodos.

Las siguientes lineas de código se encargan de actualizar la mejor solución hasta ahora y también lo necesario para el siguiente paso. Luego, la complejidad de este paso es $O(n^3)$.

Paso 3:

Funciones que utilizo:

- *sonAdyacentes* > Indica si dos nodos son adyacentes en el grafo (O(1))
- *noEstabaEnClique* > Chequea que el nodo en el que estoy parado actualmente no esté ya en la clique viendo si *vecSolIte[i]* es 0 o 1. ($O(1)$)
- *nodoFormaClique* > Analiza si el nodo i forma una clique con todos los nodos que ya tenia anteriormente. Tiene complejidad $O(n)$ y el pseudocódigo será especificado más adelante.

A su vez, utilizo la función *calcFrontera* ya explicada anteriormente que tiene complejidad $O(n^2)$.

Por otro lado, cada ciclo for tiene complejidad $O(n^3)$ pues la operación más costosa que tiene adentro es $O(n^2)$ y el mismo itera n veces.

Finalmente, vemos que el ciclo while por un lado no puede iterar más de n veces pues en cada iteración se agrega un nodo a la clique y luego de n iteraciones nos quedaríamos sin nodos nuevos, y por otro lado en el peor caso itera $O(n)$ veces ya que si tenemos un grafo completo, comenzando del nodo de mayor grado, si vamos extendiendo la clique nodo a nodo vamos a poder llegar al último nodo del grafo generando siempre cliques más grandes.

Por esto, el *paso 3* tiene una complejidad teórica de $O(n^4)$.

Complejidad teórica final:

Habiendo calculado la complejidad de cada uno de los pasos de la heurística puedo decir que finalmente, la complejidad teórica es de $O(n^2 + n^3 + n^4) = O(n^4)$.

Observación. La operación *calcFrontera* se pudiese haber modificado para realizarse en O(n), ya que estando en la iteración i del ciclo while y teniendo guardada la frontera de $i-1$ nodos, al agregar un nodo v en esa iteración la nueva frontera sería: *Frontera*($i-1$ nodos) + *gradoNodo*(v) − ($i-1$). En otras palabras, es sumarle a la frontera anterior el grado del nodo agregado restándole las aristas que tiene hacia nodos de la clique (*i - 1*). Con la modificación descripta, la complejidad final sería O(n^3).

Sin embargo, como la próxima heurística golosa que presentaremos tiene complejidad teórica cúbica, se decidió que *MFCGM* sea de orden O(n^4) para enriquecer la experimentación.

Algorithm 7 agregarVecinoMejorFrontera

```
procedure AGREGARVECINOMEJORFRONTERA(Graph grafo, vector<int> vecAux, vector<int> vecSolIte, int posFrontMejor, int nodoGM, int ultNodoAg)
    for int i = 0 to grafo→n do                                          ▷ O(n^3)
        if i ≠ nodoGM ∧ sonAdyacentes(grafo, nodoGM, i) then             ▷ O(n^2)
            vecAux[i] = 1                                                ▷ O(1)
            aux ← calcFrontera(grafo, vecAux)                            ▷ O(n^2)
            if aux > posFrontMejor then                                  ▷ O(n)
                vecSolIte = vecAux                                       ▷ O(n)
                posFrontMejor = aux                                      ▷ O(1)
                ultNodoAg = i                                            ▷ O(1)
            vecAux[i] = 0                                                ▷ O(1)
```

Algorithm 8 nodoFormaClique

```
procedure NODOFORMACLIQUE(Graph grafo, vector<int vec, int nodo)
    res ← true
    for int i = 0 to grafo→n do                                          ▷ O(n)
        if i ≠ nodo ∧ vec[i] = 1 ∧ grafo→matrizAdy[nodo][i] = 0 then
            res = false
    return res
```

3.2 Clique Maxima

La heurística *CM* -máxima clique mayor frontera- se basa en suponer que la clique máxima del grafo tiene mas probabilidades de ser la de mayor frontera. En caso de encontrar varias cliques maximas del mismo tamaño, el algoritmo se queda con la de mayor frontera. Esto permite solucionar correctamente el peor caso de la heurística *MFCGM*.

Al intentar buscar la clique máxima, nos encontramos frente a otro problema NP-Completo. Entonces debemos aplicar otra heurística golosa para resolver el problema. Esta se va a basar en ordenar el arreglo de nodos en orden descendente respecto al grado del mismo, y tomando cada nodo como inicial, ir iterando sobre el arreglo en el orden mencionado, con el fin de formar su clique maximal. Nos vamos guardando la clique maxima hasta ahora conseguida, y comparando con la clique de iteración. En caso de ser de igual tamaño, se reemplaza en base a la clique de mayor frontera. Luego, devolvemos la solución.

En detalle, los pasos a seguir serian:

- **Paso 1**: Crear un nuevo arreglo con los nodos del grafo.
- **Paso 2**: Ordenar de forma descendiente respecto al mayor grado. Utilizamos Selection Sort.
- **Paso 3**: Crear clique solucion parcial.
- **Paso 4**: Tomar el primer nodo del arreglo como nodo inicial y agregarlo a la clique. Ir iterando sobre el arreglo, agregando (siempre que sea posible) el nodo a la clique solución parcial.
- **Paso 5**: Terminada la iteración, comparar la clique solución parcial con la clique solución final guardada anteriormente. En caso de que la parcial sea mayor, reemplazar clique final por clique parcial. De ser de igual tamaño, reemplazar por la de mayor frontera.
- **Paso 6**: Repetir desde Paso 3, pero esta vez tomando como nodo inicial el siguiente nodo del que fue tomando anteriormente.
- **Paso 7**: Una vez calculadas todas las cliques parciales, utilizando todos los nodos como nodo inicial, devolver clique máxima final. Su frontera es la solución.

Al finalizar estos pasos tengo una solución que se generó de forma golosa con un algoritmo de complejidad polinomial (se analizará más adelante).

3.2.1 Instancias no optimas

Las instancias de peor caso para esta heurìstica son los grafos completos.

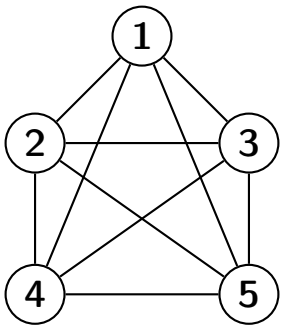


Figure 4: Instancia Completa K_5

En estas instancias, clique maxima es exactamente *todo* el grafo. Al estar todas las aristas incluidas dentro de el, la frontera maxima obtenida serà siempre 0. Dado esto, cualquier subgrafo incluido en el, (incluyendo un nodo no aislado) conformará una clique de mayor frontera que la soluciòn devuelta. Para peor, nuestra heurística de búsqueda de clique máxima **nunca va a fallar en estos casos**, y siempre devolverá como solución la clique igual a todo el grafo.

Recordemos la forma de obtener la clique máxima implementada por nuestra heurística con el **Paso 4**. Ahora, sin importar el orden en que esten ordenados los nodos (ya que todos tienen el mismo grado), partiendo de cualquier nodo raiz, se podrá agregar a la clique el nodo por el que se está iterando. Devolviendo como resultado de mayor clique todo el grafo.

Luego de observar el comportamiento de la heurística en grafos completos, vamos a utilizar nuestra intuición para formular la siguiente hipótesis: Entre más sea la densidad del grafo, mas lejos esta la heurística de devolver el resultado

óptimo. De ahí a que los grafos con mayor densidad (grafo completo, densidad = 1) tengan los peores resultados. Veremos la experimentación correspondiente en la sección **3.3**

3.2.2 PseudoCódigo y Complejidad teórica

Algorithm 9 CM

```
procedure CM(Graph grafo)
    res ← 0                                              ▷ O(1)
    vecRes ← 0                                           ▷ O(N)
    CMaux(grafo, res, vecRes)                            ▷ O(N^3)
    return (res, vecRes)
```

Algorithm 10 CMaux

```
procedure CMAUX(Graph grafo, int res, vector<int> vecRes)
    cantNodosG ← grafo.n
    NodosOrdenados[cantNodosG]                           ▷ O(1)
    Inicializar(NodosOrdenados)                          ▷ O(N)
    OrdenarPorGrado(NodosOrdenados)                      ▷ O(N^3)

    for i = 0 to cantNodosG do                           ▷ O(N)
        vecPasos(cantNodosG, 0)
        vecPasos[NodosOrdenados[i]] = 1
        for j = 0 to cantNodosG do                       ▷ O(N)
            if i != j ∧ nodoFormaClique(grafo, vecPasos, NodosOrdenados[j]) then   ▷ O(N)
                vecPasos[NodosOrdenados[j] = 1;
        if cantNodos(vecRes) < cantNodos(vecPasos) then  ▷ O(N)
            vecRes = vecPasos
        else
            if cantNodos(vecRes) == cantNodos(vecPasos) then   ▷ O(N)
                if calcFrontera(grafo, vecPasos) > calcFrontera(grafo, vecRes) then   ▷ O(N^2)
                    vecRes = vecPasos

    res = calcFrontera(grafo, vecRes)                    ▷ O(N^2)
```

En el **Algoritmo 10** se pueden divisar los pasos que describimos en la **sección 3.2**, donde el *Paso 1* y *Paso 2* consiste en las lineas de código antes del primer **For**, el ciclo de pasos, desde el *Paso 3* hasta el *Paso 6* incluido, comienza en el ciclo y sigue termina en la última linea de código, cuando devolvemos el resultado. Esta última linea es el *Paso 7*. Analizaremos la complejidad de cada uno de estos pasos por separado (con alguna explicación para aclarar más el pseudocódigo en algunos casos). Finalmente, la complejidad teórica final será $O(complej(Paso1 y Paso2) + complej(Paso3 al Paso6) + complej(Paso7))$

Paso 1 y Paso 2:

Primero, y para una mayor facilidad, guardamos la cantidad de nodos del grafo en la variable *cantNodosG*, en tiempo constante.

Luego procedemos a crear un arreglo de tamaño *cantNodosG*, y ni bien creado, le aplicamos *Inicializar*. De esta manera, instanciamos el valor de cada elemento del arreglo igual a su posición. O sea, $NodosOrdenados[i] = i$. Lo vamos a necesitar para identificar inequívocamente a cada nodo dentro del arreglo. Su complejidad temporal es $O(2n) = O(n)$

Finalmente ordenamos el arreglo por grado con la función *OrdenarPorGrado*. El algoritmo de ordenamiento aplicado es un **Selection Sort**. Su complejidad seria $O(n^2)$, sin embargo, en la comparación entre nodos en cada iteración se utiliza la funcion *gradoNodo*, que calcula el grado del nodo a comparar, con una complejidad de $O(n)$. Entonces, la complejidad total de *OrdenarPorGrado* es $O(n^2 * n) = O(n^3)$

Finalmente, la complejidad total de **Paso 1 y Paso 2** es $O(1 + n + n^3) = O(n^3)$.

Paso 3 al Paso 6:

Entramos en el primer ciclo, y para cada iteración hacemos:

- **Creo una nueva Clique parcial:** $O(n)$
- **Marco a mi nodo actual de la iteracion dentro de la clique:** $O(1)$
- **Un nuevo ciclo sobre el arreglo:** Voy a ir agregando, de ser posible, cada nodo a la clique, teniendo como nodo base el asignado en el primer ciclo. Utilizo la funcion *NodoFormaClique*. Complejidad total del ciclo: $O(n^2)$
- **Comparo Cliques:** Primero comparo tamaños, y reemplazo por la mas grande, esto es $O(n)$. En caso de tener el mismo tamaño, calculo sus fronteras y reemplazo por la de mayor valor. Utilizo la funcion *calcFrontera*, cuya complejidad es $O(n^2)$. Complejidad total de comparar cliques: $O(n + n^2) = O(n^2)$

Las funciones auxiliares utilizadas son *calcFrontera* y *NodoFormaClique*. Cuyos pseudocódigos corresponden al **Algoritmo 4** y **Algoritmo 8** respectivamente.
Finalmente, la complejidad de **Paso 3 al Paso 6** es $O(n * (n + 1 + n^2 + n^2)) = O(n^3)$

Paso 7:

Solo hay que calcular la frontera de la clique solución que encontramos. Volvemos a utilizar la función *calcFrontera*. Esto es $O(n^2)$

Complejidad teórica final::

Habiendo calculado la complejidad de cada uno de los pasos de la heurística, tenemos finalmente que la complejidad es $O(n^3 + n^3 + n^2) = O(n^3)$

3.3 Experimentación

En esta sección realizamos diversas experimentaciones:

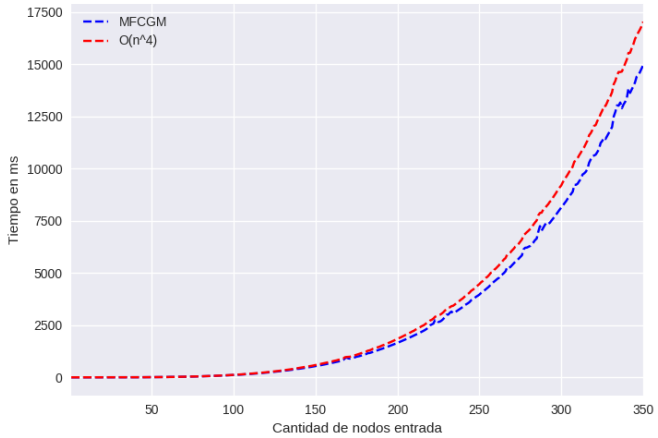
- Corremos ambas heurísticas con grafos completos (donde en particular *MFCGM* debiera tener un tiempo alto de cómputo) y comparamos sus tiempos de cómputo con las curvas de funciones $O(n^3)$ y $O(n^4)$ para verificar que se cumplen las complejidades.
- Comparamos las instancias no óptimas (aquellas que devuelven siempre una frontera mala) de cada heurística con lo que debiera ser la respuesta correcta.
- Comparamos la frontera obtenida y el tiempo de cómputo de cada heurística con el algoritmo exacto en grafos generados al azar según cada cantidad de nodos, fijando cierta densidad de aristas.
- Experimentación general para grafos con cantidad de nodos constante y densidad variable. Comparando el algoritmo exacto y la heurística **CM**.
- Experimentación general para grafos al azar hasta n grande con cierta densidad fija para ver qué heurística devuelve una frontera mejor en general.

3.3.1 Justificación de complejidades teóricas

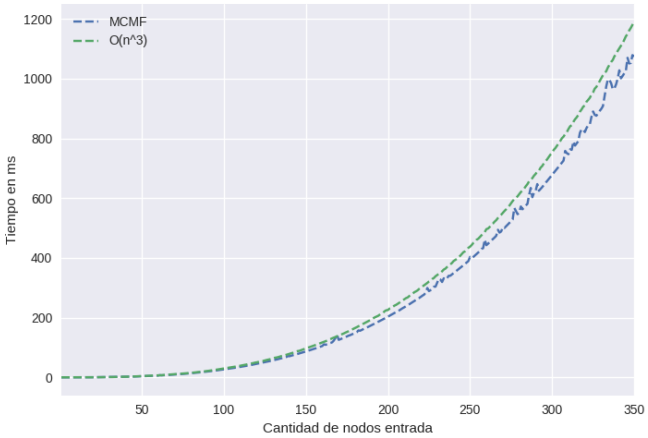
Fijando grafos completos como entrada, sea n la cantidad de nodos. Para cada n desde 1 hasta 350 se corrieron 40 repeticiones de cada heurística y se tomó el promedio de los tiempos de cómputo según cada n.

A su vez, se crearon funciones que representen correctamente las complejidades $O(n^3)$ y $O(n^4)$ -3 o 4 ciclos anidados que se recorran n veces- y para éstas también se tomó el promedio de 40 repeticiones para cada n.

Se eligieron grafos completos para esta experimentación pues en ellos se ve bien reflejada la complejidad de *MFCGM* pues el ciclo while se recorre O(n) veces.



(a) Figure 4: MFCGM vs $O(n^4)$



(b) Figure 5: CM vs $O(n^3)$

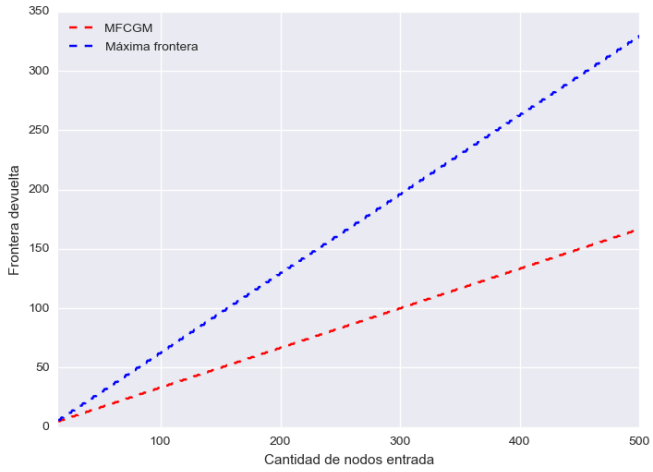
Analizando las **Figuras 4 y 5** podemos ver como en cada caso las curvas generadas por la ejecución de las heurísticas son similares a las curvas de las funciones que representan su complejidad. Luego, concluímos que las complejidades teóricas de cada una de las heurísticas se cumplen.

3.3.2 Instancias no óptimas de cada heurística

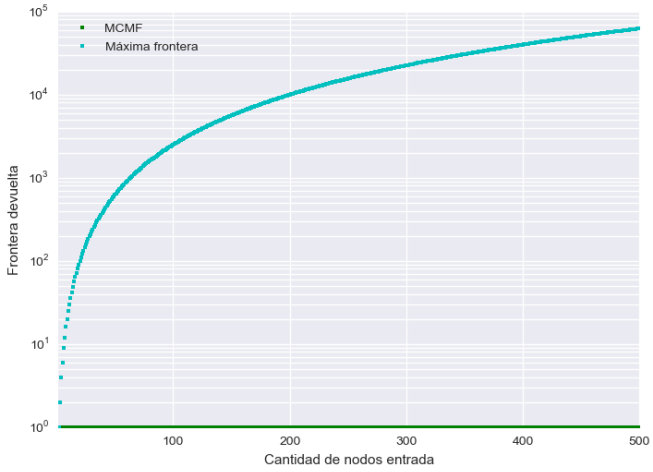
En esta subsección vamos a comprobar que las instancias no óptimas para cada una de las heurísticas explicadas previamente, efectivamente se comportan como lo expusimos. Recordemos que estas instancias eran:

- **MFCGM.** Familia de grafos nombrada como 'Estrella no óptima' que tenía dos componentes conexas. Una de ellas tenía al nodo de mayor grado, y la otra tenía la frontera máxima.
- **CM.** Grafos completos.

Sea n la cantidad de nodos, para cada n entre 1 y 500 se corrió la heurística MFCGM con la instancia *Estrella no óptima*, y por otro lado se graficó la curva que corresponde a la máxima frontera real para cada n (o sea, lo que devolvería el algoritmo exacto). Análogamente se realizó el mismo proceso con la heurística *CM* en relación a grafos completos.



(a) Figure 6: Frontera máxima vs MFCGM - *Estrella no óptima*



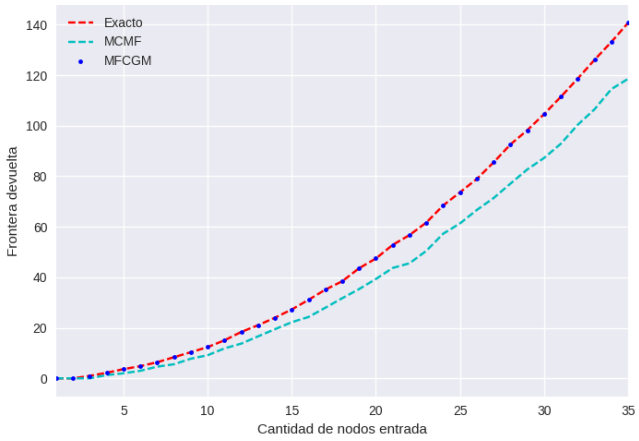
(b) Figure 7: Frontera máxima vs CM - *Grafos Completos*

Observando las **Figuras 6 y 7** vemos lo siguiente:

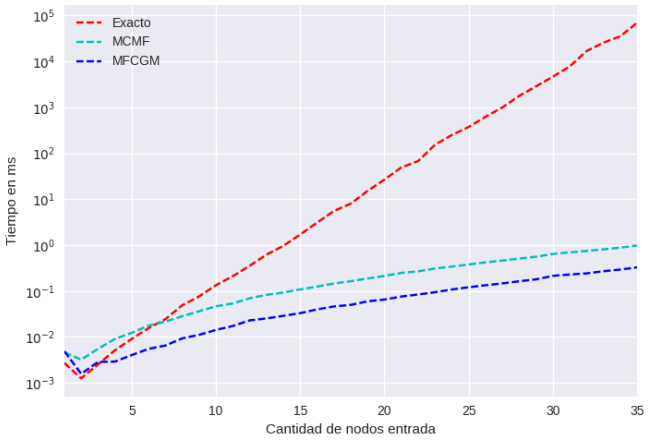
- El resultado exacto de las fronteras máximas aumenta bastante más rápido que el resultado que devuelve la heurística *MFCGM* con la familia de grafos *Estrella no óptima* como entrada, comprobando la hipótesis propuesta en la sección **3.1.1**
- Tal cual fue expuesto previamente, *CM* devuelve 0 si el grafo de entrada es completo. Para que sea más notorio que el resultado es 0, se decidió sumarle 1 y así poder graficarlo en escala logarítmica. De lo contrario, la frontera devuelta crecía tanto que la curva correspondiente a *MFMF* quedaba siempre en 0, cuando en realidad podría haber estado creciendo pero igualmente no se iba a notar en el gráfico.

3.3.3 Heurística 1 vs Heurística 2 vs Algoritmo Exacto

Para estos tests generamos instancias al azar. Sea n la cantidad de nodos, para n entre 1 y 35 creamos 40 instancias de forma aleatoria con un generador de grafos hecho en C++, que recibe como parámetros la cantidad de nodos y la densidad de aristas que querés y genera el grafo. Luego, en ésta experimentación todo grafo tendrá una densidad de aristas de $0,5$. Para cada n se tomó el resultado y tiempo promedio de todas las instancias con n nodos con respecto a cada uno de los algoritmos. La finalidad del test es exponer que efectivamente las heurísticas no devuelven siempre la solución exacta, y también que quede claro que obtener la solución exacta es muchísimo más costoso temporalmente.



(a) Figure 8: Heurísticas vs Exacto Grafos aleatorios resultado



(b) Figure 9: Heurísticas vs Exacto Grafos aleatorios tiempo

Vemos en la **Figura 9** que los tiempos de cómputo no nos dieron ninguna sorpresa. El algoritmo exacto tiene un crecimiento de carácter exponencial mientras que las heurísticas no. Si bien la heurística *MFCGM* pareciera tener tiempos menores a *CM* siendo de mayor complejidad teórica, no le damos importancia a éste fenómeno pues son cantidades de nodos pequeñas que no sirven para el análisis temporal profundo. Por otro lado, yaa justificamos las complejidades teóricas de las heurísticas en el punto **3.3.1**.

Sin embargo, la **Figura 8** muestra incongruencias, ya que a priori esperaríamos que las heurísticas devuelvan peores soluciones que el algoritmo exacto, y en este caso *MFCGM* devuelve soluciones igual de buenas. De todas formas, no es muy raro que grafos generados al azar que tengan cantidades de nodos pequeñas la heurística *MFCGM* devuelva la solución correcta pues en éstos es bastante probable que el nodo de mayor grado pertenezca a la clique de mayor frontera (lo que hace la heurística devuelva la solución correcta). Por otro lado, observando más de cerca los resultados que obtuvimos en el archivo csv que fue utilizado para armar el gráfico, extraemos lo siguiente:

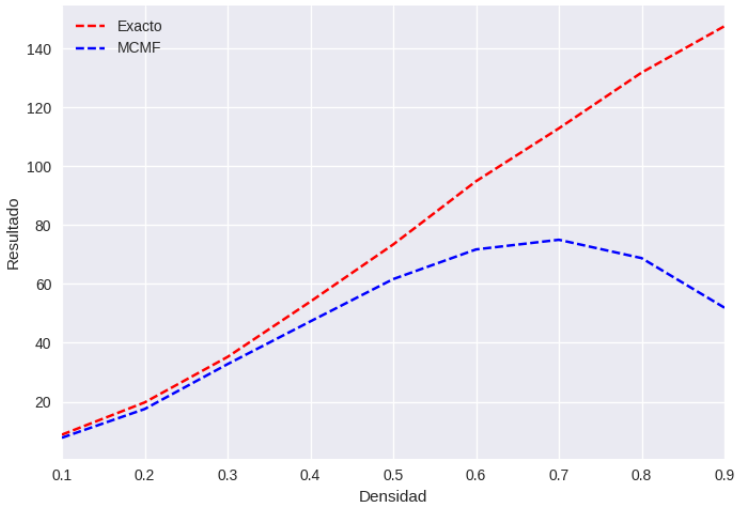
cantNod	Resultado	TiempoMS	Algoritmo
35	126	0.2192	MFCGM
35	127	106240	Exacto

Donde ambas lineas corresponden a la ejecución del mismo grafo.

Este resultado no se ve reflejado en el gráfico pues se tomó el promedio de las 40 instancias de cada n y que haya un sólo caso donde el resultado difiere (y por una diferencia de 1) no afecta al promedio.

3.3.4 Variación de densidades en CM

En la sección **3.2.1**, luego de mostrar la peor instancia para la heurística **CM** (ya demostrado en la sección **3.3.2**), se planteó la hipótesis de que a mayor densidad del grafo, peor resultado de la heurística obtendríamos. Para esto se fijo una cantidad de nodos (utilizamos 25. Un numero chico para que el algoritmo exacto termine en tiempos razonables), y variando en la densidad de los grafos, aleatorizando la unión entre nodos con 25 iteraciones por densidad, midieron los resultados del algoritmo exacto y la heurística CM.

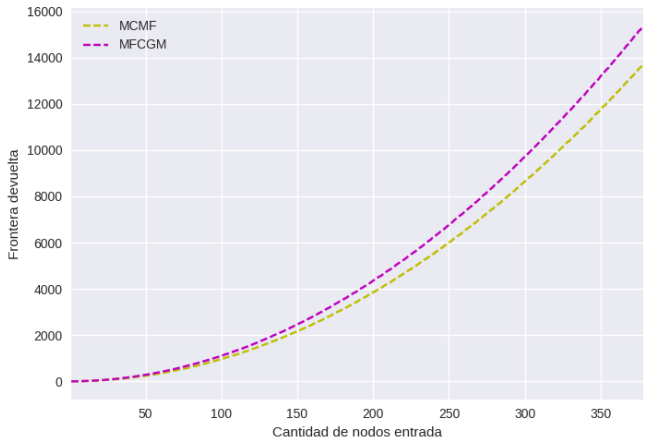


(a) Figure 10: Resultados grafos nodo fijo, densidad variable.

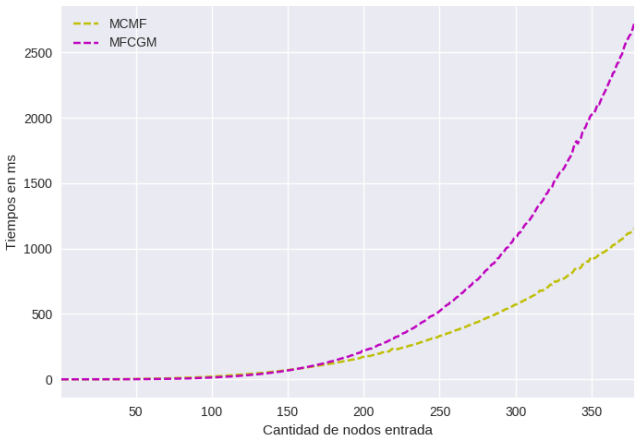
Al ver los resultados, verificamos nuestra hipótesis. La brecha de diferencia entre el resultado de la heurística y el algoritmo exacto es mas amplia a medida que se aumenta la densidad del grafo. También podemos observar como nuestra heurística se comporta como una loma. Llega a un punto máximo (aproximadamente por la densidad 0.7) y luego empieza a devolver resultados peores a medida que crece la densidad. Esto es interesante y complementa con la hipótesis original, que muestra que al llegar a la densidad completa, tendremos el peor caso, retornando siempre 0

3.3.5 Experimentación general

Aquí también utilizamos, como en el test anterior, el generador de grafos. Fijamos una densidad de $0,5$ y experimentamos para n entre 1 y 250 ambas heurísticas con 40 instancias aleatorias para cada n (y tomando el promedio tanto en resultado como en tiempo). El objetivo de ésta experimentación es definir qué heurística nos resulta mejor en general.



(a) Figure 11: Resultado heurísticas Grafos aleatorios



(b) Figure 12: Tiempo heurísticas Grafos aleatorios

Aquí los resultados fueron como esperábamos. La heurística *MFCGM* devuelve resultados mejores que la heurística *CM* pero también tiene un tiempo de cómputo que, a medida que incrementa la cantidad de nodos, crece de forma muy veloz.

Concluimos que si buscamos una solución para una instancia muy grande de forma rápida y no nos importa tanto lo buena que sea la misma, entonces nos quedamos con la heurística *CM*. De lo contrario, si uno está dispuesto a dejar de lado el tiempo de cómputo porque prefiere que la solución sea mejor, entonces debe utilizar la heurística *MFCGM*.

4 Heurística Búsqueda Local

En esta sección se explica la heurística de búsqueda local utilizada. La misma se basa en:

Dada una solución posible al problema, definir un conjunto de soluciones vecinas de magnitud polinomial. Luego buscar cuál solución es mejor entre todas las vecinas y volver a aplicarle el algoritmo a ésta (hasta que no haya una mejor). En otras palabras básicamente vamos yendo entre soluciones vecinas hasta encontrar alguna que sea la mejor localmente, por lo que no me convendrá seguir por algún vecino.

Es importante que el conjunto definido para una solución sea de magnitud polinomial ya que justamente en una heurística tratamos de resolver problemas cuya solución exacta es de carácter exponencial. Si nosotros no acotamos a éstos términos el conjunto de vecinos, no habría punto en ejecutar esta heurística.

Lo positivo de la búsqueda local es que se puede combinar con otras heurísticas ya planteadas para mejorar sus soluciones, por ejemplo dada una heurística golosa que te devuelve una solución posible, se empieza el algoritmo de búsqueda local a partir de este resultado y finalmente vamos a terminar encontrando un resultado mejor o igual al recibido.

Como lado negativo, obviamente como toda heurística no devuelve un resultado exacto. El algoritmo como describimos se basa en encontrar algún pozo dentro de las soluciones, pero no en encontrar el mejor de todos. Si en alguna iteración sucede que encontramos un máximo local -pero no máximo global- el algoritmo va a parar su búsqueda y retornar un resultado no óptimo.

En lo que respecta a nuestro algoritmo de búsqueda local definimos dos diferentes heurísticas, las cuáles difieren en la cardinalidad del conjunto de vecinos para cada solución. La que vamos a llamar Vecindad Lineal define un conjunto de vecinos de orden $O(n)$ para cada solución posible, mientras que la que vamos a llamar Vecindad Cuadrática toma un conjunto de vecinos de orden $O(n^2)$. Vamos a explicar cada algoritmo y sus diferencias en las siguientes secciones.

4.1 Vecindad Lineal

En nuestra primer heurística de búsqueda local definimos, como dijimos previamente, una vecindad O(n) (con n la cantidad de nodos del grafo). Dados dos conjuntos soluciones de nuestro problema S y S', éstas son vecinas si y sólo si se cumple que, sea C el conjunto resultante de la operación $V(S)/V(S')$, entonces $\#C = 1$.

Básicamente se obtiene que las soluciones vecinas de un conjunto S son aquellas de la pinta $S + x$ -siendo x un nodo perteneciente a G que no pertenecía a S y sigue cumpliendo el invariante de una solución- o S - x, siendo x un nodo perteneciente a S.

De esta manera obtenemos a los sumo n posibles vecinos de S ya que el cardinal del conjunto unión entre $NodoFueraDeS$ y $NodosDentroDeS$ es igual a n.

En nuestro algoritmo vamos a partir desde una solución semilla pasada por parámetro, y comenzar las iteraciones del algoritmo frenando cuando nos paremos sobre una solución que sea la mejor entre todas las vecinas definidas anteriormente.

Dentro de cada iteración lo que hacemos es separar en dos conjuntos: Por un lado todos los nodos x de G que puedo agregar a la solución actual S (manteniendo que $S + x$ siga siendo una clique de G), y por otro una copia de S, cosa de poder iterar los elementos que le voy a poder restar a S.

Para generar el primer conjunto ejecutamos una iteración sobre todos los nodos de G agregando a los elementos que cumplan:

- $x \notin S$
- $S \subset VecinosDe(x)$

Ya armados estos dos conjuntos, simplemente calculamos cuántas aristas fronterizas tiene la solución generada tanto de sumar o restar a cada nodo, y luego nos quedamos con la mejor estricta en caso de haberla, y comenzamos una nueva iteración del algoritmo a partir de esta nueva solución. En caso de no encontrar una solución vecina que mejore a la actual, el algoritmo se frena ya que entramos en un pozo.

Los puntos positivos de la Vecindad Lineal es que obtener los vecinos es una tarea muy simple, y la complejidad incluso en el peor caso no llega a ser muy elevada, como vamos a ver posteriormente.

Sin embargo puede pasar que ésta búsqueda se desvíe del camino hacia la mejor solución, en caso de que ésta tenga la misma o menor cardinalidad que la actual, por tener que pasar primero por la mejor solución, entre las que

se encuentra de sacarle un nodo a S. Para tratar de evitar este problema buscamos un algoritmo cuyo conjunto de vecinos considere soluciones un poco más allá que la inmediata modificación de S, e implementamos para ellos la heurística Vecindad Cuadrática.

4.2 Vecindad Cuadrática

Tanto la idea como la implementación de esta heurística son muy similares a la previamente vista. En lo que se diferencian simplemente es que el conjunto de vecinos por cada solución S del problema aumenta.

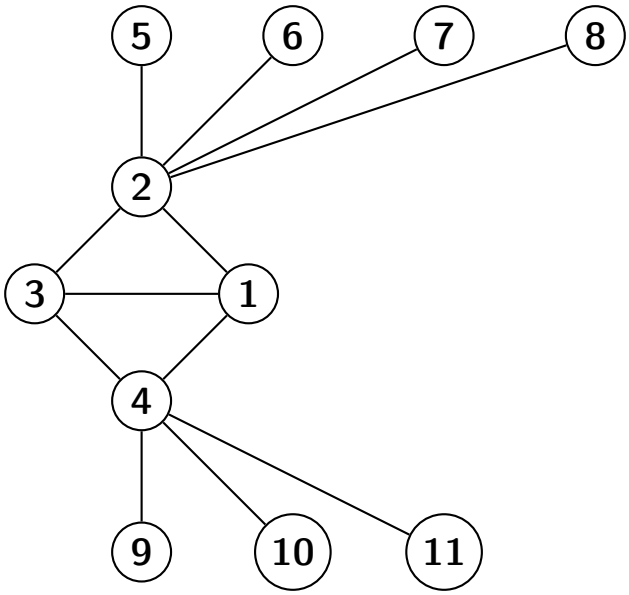
Con esta vecindad dos soluciones S y S' son vecinas entre sí, si cumplen una de las siguientes condiciones:

- El Cardinal del conjunto resultante de la operación $V(S)/V(S')$ es igual a 1
- Existe $v' \in S'$ tal que para algún vértice $v \in S$, se tiene que $(S - v) + v' = S'$

Éste conjunto de vecinos tiene cardinalidad de orden cuadrático con respecto a los nodos del grafo, porque en el peor caso, cuando G es un grafo completo, pasa que para cada nodo v dentro de S, v puede ser swapeado por cualquiera de los nodos que no se encuentran ya en S, logrando sumar n vecinos por cada vértice dentro de S y una complejidad espacial de orden $O(n^2)$

En cuanto a su implementación algorítmica no suma ninguna complejidad agregar esta condición, puesto que sólo hay que considerar un nuevo conjunto de soluciones posibles a las cuáles moverse, pero la idea general permanece intacta.

Esta idea surgió para evitar casos como el siguiente:



Aquí si la solución semilla es el K_3 formado por los nodos $[1, 3, 4]$, al ejecutar la Búsqueda Local definida con vecindad Lineal hubiese devuelto la clique formada por los nodos $[1, 4]$ con 6 aristas fronterizas. Mientras que la búsqueda Local definida con Vecindad Cuadrática hubiese considerado el caso de eliminar el nodo 4 de la solución y agregar al nodo 2, el cual mejoraba el escenario. Y luego de ahí restar el nodo tres, dando como resultado final a la clique $[1, 2]$ con un total de 7 aristas fronterizas.

4.3 Implementación

4.3.1 Vecindad Lineal

Para implementar el algoritmo elegimos una estructura levemente diferente a las vistas en la materia. Utilizamos una representación de grafos de Arreglos de árboles logarítmicos (estructura *Set* de c++).

El por qué de esta elección está explicado más adelante pero para ello primero tenemos que mostrar el pseudo código de nuestra heurística:

```
1: procedure BusquedaLocalLineal(In G: Arreglo<Conj<int», In/Out SolAct: Conj<int>)→int
2:     bool NoEncontréPozo = true
3:     while NoEncontréPozo do
4:         NodosQuePuedoSumar = new Conj<int>()                                   ▷ O(1)
5:         NodosQuePuedoRestar = new Conj<int>()                                  ▷ O(1)
6:         NodoQueMejoraAlSumar = <-1,-1>   ▷ Tupla. First representa el nodo, y Second las fronteras que se
   tienen al sumarlo
7:         NodoQueMejoraAlRestar = <-1,-1>   ▷ Tupla. First representa al nodo, y Second las fronteras que se
   tienen al restarlo
8:                                                  ▷ Lleno el conjunto NodosQuePuedoSumar
9:         for i in [0...n) do
10:            if NodoNoEstaEnSolucionActual(SolAct, i) then                       ▷ O(log(n))
11:                bool EsVecinoDeTodos = true                                     ▷ O(1)
12:                for j in SolAct do
13:                    if ¬SonVecinos(j, i) then                                   ▷ O(log(m))
14:                        EsVecinoDeTodos = false                                 ▷ O(1)
15:                endFor   ▷ Como itera |SolAct| veces con una operación de O(log(m)), termina con complejidad
   O(n ∗ log(m))
16:                if EsVecinoDeTodos then                                         ▷ O(1)
17:                    NodosQuePuedoSumar.Add(i)                                   ▷ O(log(n))
18:        endFor                                                                  ▷ O(n² ∗ log(m))
                                                    ▷ Lleno el conjunto de NodosQuePuedoRestar
19:        for i in SolAct do
20:            NodosQuePuedoRestar.Add(i)                                          ▷ O(log(n))
21:        endFor                                                                  ▷ O(n ∗ log(n))
22:                                          ▷ Calculamos las aristas fronterizas entre todos los vecinos
23:        int CantidadDeFronterasActual = CalcularFronteras(G, SolAct)            ▷ O(n ∗ log(n) ∗ m)
24:        for i in NodosQuePuedoSumar do
25:            SolAct.Add(i)                                                       ▷ O(log(n))
26:            CantFronterasParcial = CalcularFronteras(G, SolAct)                 ▷ O(n ∗ log(n) ∗ m)
27:            if MejoroLaSolucionQueTeniaDeSumarNodos(NodoQueMejoraAlSumar, CantFronterasParcial)
   then                                                                            ▷ O(1)
28:                NodoQueMejoraAlSumar = <i, CantFronterasParcial>                ▷ O(1)
29:            SolAct.Erase(i)                                                     ▷ O(log(n))
30:        endFor                                                                  ▷ O(n² ∗ log(n) ∗ m)
31:        for i in NodosQuePuedoRestar do
32:            SolAct.Erase(i)
33:            CantFronterasParcial = CalcularFronteras(G, SolAct)
34:            if MejoroLaSolucionQueTeniaDeRestarNodos(NodoQueMejoraAlRestar, CantFronterasParcial)
   then
35:                NodoQueMejoraAlRestar = <i, CantFronterasParcial>
36:            SolAct.Add(i)
37:        endFor                                                                  ▷ O(n² ∗ log(n) ∗ m)
38:                                          ▷ Ahora queda simplemente ver Cual es la mejor
39:        if SiNingunaFronteraSuperaALaActual then                                ▷ O(1)
40:            NoEncontrePozo = false                                              ▷ O(1)
41:        else if SiLaMejorFronteraEsSumandoUnNodo then                           ▷ O(1)
42:            SolAct.Add(NodoQueMejoraSuma.First)                                 ▷ O(log(n)
43:        else
44:            SolAct.Erase(NodoQUeMejoraRestar.First)                             ▷ O(log(n))
45:    endWhile
46:    return CalcularFronteras(G,SolAct)
```

```
1: procedure CalcularFronteras(In G: Arreglo<Conj<int», In/Out SolAct: Conj<int>)→int
2:     res = 0
3:     for i in Sol do                                                    ▷ O(n)
4:         for j in Vecinos(i) do                                         ▷ O(m)
5:             if ¬ PerteneceA(j,Sol) then                                ▷ O(log(n))
6:                 res +=1
7:         endFor
8:     endFor                                                             ▷ O(n * m * log(n))
```

Entonces como podemos ver dentro de cada iteración ejecutamos varias operaciones de complejidad temporal alta, por lo que buscamos una estructura que logre optimizar la complejidad total. Para ello hicimos el siguiente análisis.

Las operaciones costosas que ejecutamos en cada iteración son:

1. Iterar todos los nodos
2. Agregar o sacar un nodo
3. Chequear que estén o no estén en mi conjunto *SolucionActual*
4. En caso de que no estén chequear si todos los elementos del conjunto *SolucionActual* son vecinos de dicho nodo, lo que implica que quiera un rápido acceso a esta respuesta
5. Calcular fronteras de un conjunto de nodos. Lo cual significa iterar sobre todos los vecinos del conjunto *SolucionActual* y preguntar para cada vecino si se encuentra o no en la misma.

Entonces planteamos el siguiente cuadro comparativo de qué costaba más dependiendo la representación

	MatrizDeAdyacencia	ArregloDeListas	ArregloDeEstructSet
Item 1	$O(n)$	$O(n)$	$O(n)$
Item 2	$O(1)$	$O(1)$	$O(log(\|Conj\|))$
Item 3	$O(\|SolAct\|)$	$O(\|SolAct\|)$	$O(log(\|SolAct\|))$
Item 4	$O(\|SolAct\| * n)$	$O(\|SolAct\| * \|Vecinos(nodo)\|)$	$O(\|SolAct\| * Log(\|Vecinos(nodo)\|))$
Item 5	$O(\|SolAct^2\| * n)$	$O(\|SolAct\|^2 * Vecinos(SolAct))$	$O(\|SolAct\| * Vecinos(SolAct) * Log(\|SolAct\|))$

Al fin de cuentas podemos ver que la representación que nos reducía mayormente la complejidad asintótica del algoritmo es la representación de un Arreglo De Sets, Donde obviamente cada nodo representa una posición del arreglo y el Set dentro el conjunto de vecinos del mismo.

La complejidad en peor caso de los algoritmos de Búsqueda Local no se puede acotar ya que éstos siguen ejecutando hasta que encuentre un pozo dentro del campo de soluciones que recorren. Sin embargo bien se puede estimar la complejidad temporal dentro de cada iteración del algoritmo, por lo que resultamos en lo siguiente:

COMPLEJIDAD POR ITERACIÓN: Como mostramos dentro del pseudocódigo la operación ejecutada de mayor complejidad dentro de cada iteración es de $O(n^2 * log(n)log(m))$, debido al ciclo que se ejecuta para buscar el mínimo entre los vecinos, ejecutando hasta n veces la operación $CalcularFronteras$ de complejidad $O(n * log(n) * m)$.

4.3.2 Vecindad Cuadrática

Su implementación es muy similar a la ya provista para la Vecindad Lineal. El pseudocódigo es el mismo, solo que agregando entre los renglones 5 - 6 lo siguiente:

```
pair<pair<int,int>, int> NodoQueMejoraRestarYSumar
conj<pair<int,int> > NodosQuePuedoRestarYSumar = new Conj<pair<int,int> >()
conj<int> NodosFueraDeSolAct = new Conj<int>();
for i in [0..n) do
    if i ∈ SolAct then                                              ▷ O(log(n))
        NodosFueraDeSolAct.Add(i);
                                                                    ▷ O(log(n) * n)
```

Luego entre los renglones 21 - 22 agregar:

```
res = 0
for i in SolAct do
    for j in NodosFueraDeSolAct do
        bool EsVecinoDeTodos = true
        for k in SolAct do
            if k ≠ i then
                if NoSonVecinos(j, k, G) then                       ▷ O(log(n))
                    EsVecinoDeTodos = false
        endFor                                                      ▷ O(n * log(n))
    endFor                                                          ▷ O(n^2 * log(n))
    if EsVecinoDeTodos then NodosQuePuedoRestarYSumar.Add(<i,k>)
endFor                                                              ▷ O(n^3 * log(n))
```

Entre los renglones 37-38:

```
for i in NodosQuePuedoRestarYSumar do
    SolaAct.Erase(i.first)
    SolAct.Insert(i.second)
    CantFronterasParcial = CalcularFronteras(G, SolAct)
    if MejoroLaSolucionEnRestarYSumar then
        NodoQueMejoraRestarYSumar.first.first = i.first
        NodoQueMejoraRestarYSumar.first.second = i.second
        NodoQueMejoraRestarYSumar.second = CantFronterasParcial
    SolaAct.Insert(i.first)
    SolAct.Erase(i.second)
endFor                                                              ▷ O(n^3 * log(n) * m)
```

Y por último entre los renglones 40 - 41:

```
if SiLaMejorFronteraEsRestandoYSumadno then
    SolAct.Insert(NodoQueMejoraRestarYSumar.first.second )          ▷ O(log(n))
    SolAct.Erase(NodoQueMejoraRestarYSumar.first.first )            ▷ O(log(n))
    NodosFueraDeSolAct.Erase(NodoQueMejoraRestarYSumar.first.second) ▷ O(log(n))
    NodosFueraDeSolAct.Insert(NodoQueMejoraRestarYSumar.first.first) ▷ O(log(n))
```

Luego podemos ver que el aumento de vecindad trae consigo un aumento en cuanto a la complejidad temporal que conlleva cada iteración de la búsqueda, dando finalmente una complejidad total de $O(n^3 * log(n) * m)$.

4.4 Experimentación

Los experimentos de esta sección se hicieron utilizando los mismos grafos que para la sección **3.3**. Cada experimento fue ejecutado evaluando los resultados con ambas vecindades con el fin de encontrar si el poder de computo que se cede al plantear una vecindad más grande logra una mejora significativa en el resultado.

En la experimentación buscamos encontrar distintas características que distingan a cada vecindad. Para ello nos fijamos su comportamiento evaluando con densidades de aristas fijas -la cual fuimos moviendo entre experimentos-. Buscamos ver cuántas iteraciones promedio ejecutaba cada algoritmo, la curva de complejidad temporal promedio que demostraba, y por último qué tan óptimo era su resultado.

En esta sección utilizamos los mismos grafos que en la experimentación de las heurísticas golosas para testear su comportamiento con entradas random.

Los tests que corrimos llegan hasta entradas más limitadas por el aumento de complejidad temporal en la ejecución de los algoritmos.

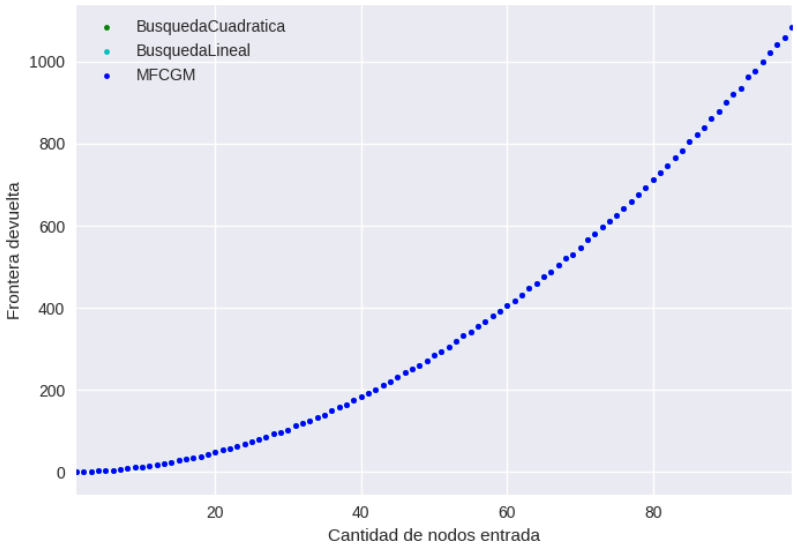
4.4.1 Experimento 1 - Resultados Devueltos

El pseudocódigo con el que corrimos los Experimentos es el siguiente:

```
1:  procedure EXPERIMENTACIÓN()
2:      for i in [5..195) do
3:          for j in [1...40] do
4:              Grafo g = GrafoRandom(i)
5:              Conj<int> Semilla1 = HeuristicaMFCGM(g)
6:              ResultadosBLLineal1 = BusquedaLineal(semilla1)
7:              ResultadosBLCuadratica1 = BusquedaCuadratica(semilla1)
8:              Conj<int> Semilla2 = HeuristicaMCMF(g)
9:              ResutladosBLLineal2 = BusquedaLineal(semila2)
10:             ResultadosBLCuadratica2 = BusquedaCuadratica(semilla2)
```

La cantidad de iteraciones con cada tamaño de N es de 40 ya que a partir de 30 casos de prueba una distribución se asemeja a la distribución normal, y tomar el promedio de ello nos devuelve resultados más fiables que con sólo 10 casos de prueba.

Búsqueda Local Implementada con Semilla de algoritmo MFCGM Sobre Grafos Random



Como podemos ver en este gráfico notamos por primera vez que los algoritmos de Búsqueda Local Lineal, Cuadrática y la Heuristica MFCGM devuelven exáctamente los mismo resultados. ¿Por qué pasa esto? Lo veremos de a partes.

1. Primero vamos a analizar la heurística MFCGM intentando descubrir por qué se da este fenómeno

El algoritmo de MFCGM, en lineas generales (como fue descripto en su sección) va agregando a la clique resultado los nodos vecinos del nodo de mayor grado, en orden del de mayor grado hacia el de menor grado, de forma que mantengan una clique entre sí y vayan mejorando la solución.

Queremos aclarar por qué afirmamos que los va agregando en orden de mayor grado a menor grado. El algoritmo verdaderamente se fija en todos los vecinos del nodo de mayor grado sin distinciones entre sí, y calcula con cada uno cuál es la frontera obtenida -si el conjunto de dos nodos era una clique-. Luego elige al que obtuvo la mayor cantidad de aristas fronterizas entre todos.

Sin embargo estas dos operaciones son equivalentes: Asumamos que tenemos dos nodos n_1 y n_2 tales que el grado de n_1 es mayor al de n_2, ambos son vecinos del nodo de máximo grado (n_m), no pertenecen a la solución actual, y la solución actual agregando n_1 o n_2 sigue siendo una clique. Entonces queremos ver que el conjunto $SolACt + n_1$ va a tener mayor cantidad de aristas fronterizas que $SolAct + n_2$.

Tenemos que la cantidad de aristas fronterizas que va a aportar cada nodo a la solución es de $g(n_x) - |SolAct|$, lo cual se deduce trivialmente porque cada arista que se conecte con un nodo dentro de $SolAct$ no va a ser fronteriza. Por lo que todo se reduce en la simple cuenta de que si se cumple la desigualdad $g(n1) > g(n2)$, por linealidad de la función resta entonces $g(n1) - |SolAct| > g(n2) - |SolAct|$, lo cuál determina que el algoritmo de MFCGM elija primero al nodo vecino de n_m de mayor grado.

Esto lo mencionamos en este momento porque mientras mirábamos los resultados de los experimentos surgió un análisis mas profundo de cómo actúa el algoritmo MFCGM, y tras este descubrimiento notamos que **el algoritmo goloso puede ser optimizado**, evitando chequear a todos los nodos vecinos de n_m dentro de cada iteración y simplemente ordenándolos al comienzo en relación a su grado. Luego ejecutando el algoritmo goloso manteniendo un índice sobre este arreglo indicando cuáles nodos ya chequeé en algún momento, y dentro de cada iteración avanzar el índice desde donde estaba.
De esta manera el chequeo por
cada nodo se ejerce una sola vez en cada ejecución y se disminuye considerablemente la complejidad teórica.

2. Ahora, con las nuevas conclusiones sobre MFCGM queremos responder por qué la heurística es un máximo local con respecto a la vecindad marcada por nuestra Vecindad Lineal.

Como dijimos anteriormente la Vecindad Lineal marca como vecinos a todas las soluciones que contengan un nodo más o un nodo menos que la solución actual. Entonces, hay que ver que MFCGM es mayor que todas las soluciones vecinas:

- *Si la solución vecina contiene un nodo menos.* Como mencionamos recientemente, la heurística golosa de la cual estamos hablando sólo agrega un nodo a su solución si cumple particularmente que la cantidad de aristas fronterizas aumentan a las del conjunto $SolAct$. Por lo que es directo decir que si le restás un nodo al conjunto solución que devuelve el algoritmo MFCGM, por invariante de cómo fue armado el conjunto, esto empeora la solución y la búsqueda local va a descartar estos casos.
- *Si la solución vecina contiene un nodo más.* Para ver este caso acudimos a cómo actúa la heurística golosa. En caso de que haya un nodo n_x tal que $SolAct + n_x$ aumenta la cantidad de fronteras y es clique del grafo, entonces sucede que en particular n_x es también vecino del n_m. Pero justamente si se daban estas dos condiciones el algoritmo de MFCGM hubiese sumado a n_x a la solución.

3. Finalmente, debemos ver por qué los resultados del algoritmo de Búsqueda Local definido con Vecindad Cuadrática no mejoran la solución provista por la heurística golosa MFCGM.

Como dijimos en secciones anteriores, la búsqueda local definida con Vecindad Cuadrática simplemente agrega a los vecinos de una solución, a aquellas soluciones del problema que se formen de restar y agregar un nodo. Luego, nos queda justificar por qué con esta nueva vecindad el resultado de MFCGM sigue permaneciendo como máximo local. Para probarlo dividiremos en dos casos:

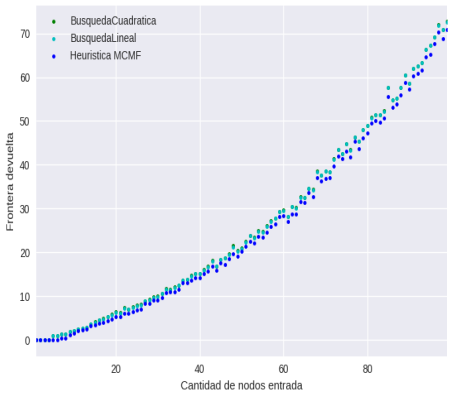
- *Si sacamos al nodo máximo n_m.* Queremos ver si suponemos que tendremos una nueva solución con frontera más grande al sacar a n_m y reemplazarlo por otro nodo diferente que sea compatible con el resto de los nodos de $SolAct$, vamos a llegar a un absurdo. Esto sale directamente con el coeficiente que planteamos anteriormente de la cantidad de fronteras que un nodo puede sumar a la $SolAct$, que era igual a $g(n) - |SolAct|$, y como tenemos que n_m es el nodo de mayor grado dentro del grafo, es evidente que no hay ningún otro nodo que eleve este coeficiente a uno más elevado que él. Por lo que el escenario de querer sacarlo no se podría de dar nunca.

- *Si sacamos cualquier otro nodo.* Aquí la explicación de por qué no hay escenario donde nos convenga sacar un nodo y agregar a otro se deriva de lo explicado en el primer punto de esta sección donde aclaramos que los nodos que se suman a la solución del algoritmo MFCGM, se suman en orden de sus grados. Por lo que todo nodo que no pertenezca a la solución y sea vecino de n_m va a tener menor o igual grado al nodo de menor grado dentro de la solución, y nuevamente llegamos a un absurdo si planteamos que se aumentan la cantidad de fronteras al swapear un nodo de mayor grado por uno de menor grado dentro de la solución.

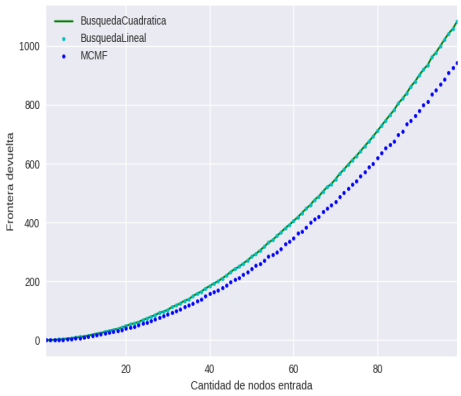
Finalmente como vemos, los resultados de aplicar búsqueda local con vecindad Lineal Y Cuadrática a la semilla provista por el algoritmo MFCGM son los esperados.

Concluimos que no hay sentido en seguir experimentando con respecto a esta heurística constructiva. Por lo que a partir de este punto hacia el resto de la experimentación, se va probar la búsqueda local siempre con la semilla provista por el algoritmo MCMF.

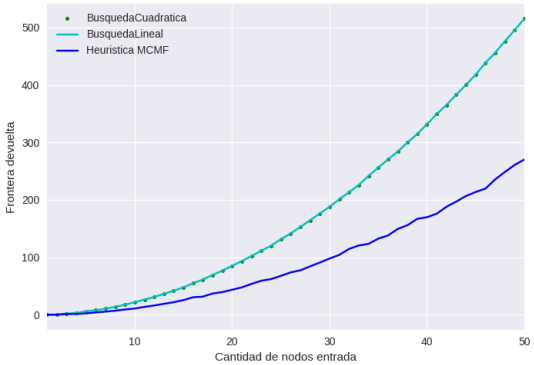
Búsqueda Local Implementada Con Semilla Heurística MCMF Sobre Grafos Random



Grafos random con Densidad 0.1



Grafos random con Densidad 0.5



Grafos random con Densidad 0.8

Como se puede observar en este experimento en primer lugar no aparecieron sorpresas mayores. Como esperábamos, la heurística de buscar la clique máxima con el fin de obtener la mayor cantidad de fronteras posibles se aleja significativamente de ser un máximo local en situaciones generales, y se ve cómo efectivamente se puede explotar el algoritmo de Búsqueda Local en estos términos. Especialmente se muestra una mejora significativa en cuanto a la solución

propuesta por el algoritmo de búsqueda local a medida que la densidad de los grafos aumenta. Esto pasa pues el goloso va encontrar cliques de mayor tamaño brindándole al algoritmo de búsqueda local más espacio para seguir su búsqueda.

Otro elemento a rescatar en esta experimentación es la similitud entre los comportamientos de las distintas Heurísticas Locales. A simple vista podrá parecer que devuelven los mismos resultados, sin embargo las diferencias entre los promedios de ejecución para cada n suele ser de no más de una frontera extra. En este punto es donde empezamos a notar que el *trade of* entre evaluar más casos, aumentando en consecuencia la complejidad del algoritmo no muestra pagar con buenos resultados.

4.4.2 Experimento 2 - Comportamiento Temporal

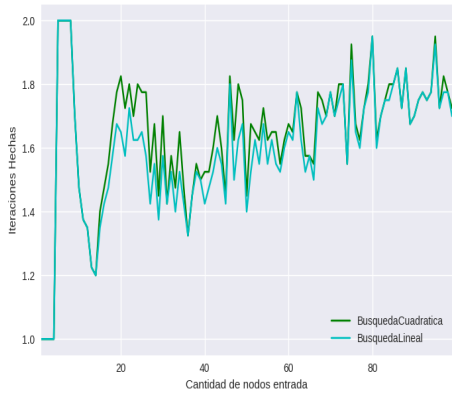
Recordamos que anteriormente en la sección de implementación mencionamos que las complejidades por iteración de cada búsqueda local eran de:

- Busqueda Lineal : $O(n^2 * log(n) * m)$
- Busqueda Cuadratica: $O(n^3 * log(n) * m)$

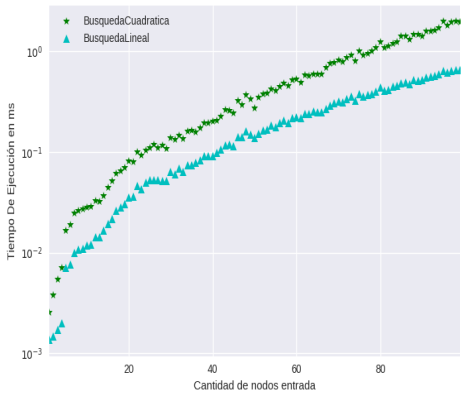
Como vemos cada complejidad depende de dos variables: la cantidad de nodos y la cantidad de aristas. Luego, experimentamos en primer lugar con los grafos randoms -para ver su complejidad temporal en los casos promedio-, y posteriormente fijando estas variables para corroborar que se cumplan los crecimientos.

Experimento Con Grafos Random

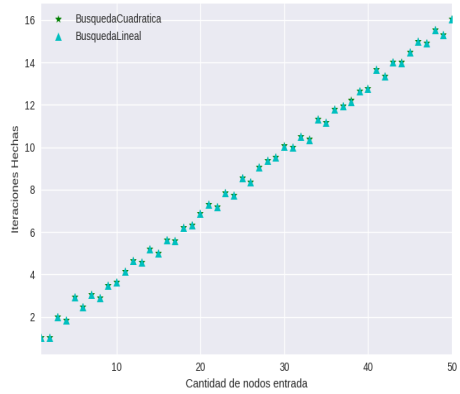
Para estas instancias se crearon grafos Random con densidad 0.1 y 0.8. Nuevamente aclaramos que éstos grafos son los mismos que fueron utilizados durante todo el trabajo práctico.



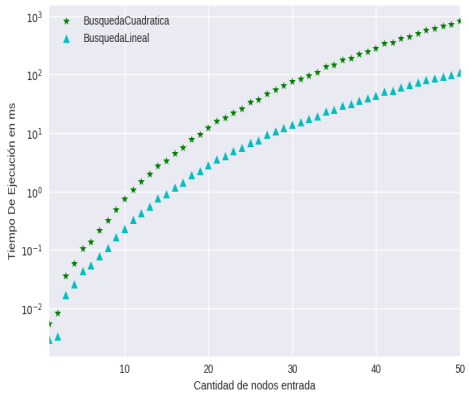
Iteraciones en Grafos random con Densidad 0.1



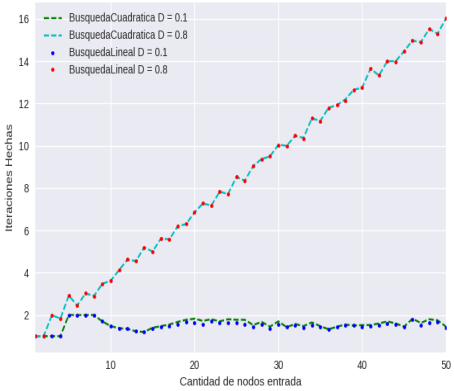
Tiempos en Grafos Random con Densidad 0.1



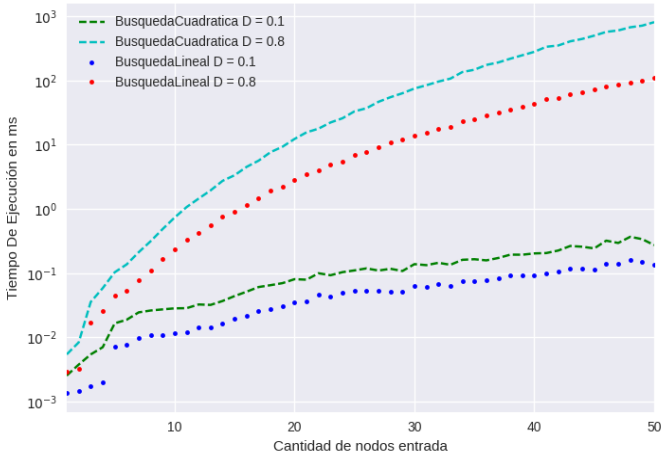
Iteraciones en Grafos random con Densidad 0.8



Tiempos en Grafos Random con Densidad 0.8



Iteraciones Densidad Alta Vs Densidad Baja



Tiempos Densidad Baja vs Densidad Alta

Como primera conclusión podemos notar como el tiempo de ejecución con grafos de baja densidad disminuye significativamente. Esto se debe a dos factores.

- En primer lugar como también se puede ver en el gráfico que compara las iteraciones entre los algoritmos que se ejecutan en grafos de densidad alta contra los que se ejecutan en grafos de densidad baja, se puede notar que la semilla de la heurística MCMF se acerca significativamente más a ser un máximo local disminuyendo en gran valor la cantidad de iteraciones que el algoritmo de búsqueda local debe ejecutar.
- Por otro lado también tenemos que gran parte de las operaciones en relación a N que se ejecutan dentro del algoritmo de búsqueda local se ejecutan por cantidad de vecinos de algún nodo. Esto implica que si la cantidad de vecinos de cada nodo disminuye, es de esperar que disminuya el tiempo de ejecución de cada iteración.

Luego ya vimos anteriormente que las diferencias entre las respuestas de ambas búsquedas locales prácticamente son nulas. Ahora también sabemos que la cantidad de iteraciones de cada vecindad no difiere sustancialmente, aunque cabe remarcar que cuando la densidad es baja se puede notar un poco más esta diferencia.

Sin embargo como vemos en los primeros dos gráficos asentados en la derecha, el precio temporal que se paga no es poco. Podemos concluir que en el caso promedio no conviene utilizar la Búsqueda local definida con vecindad Cuadrática.

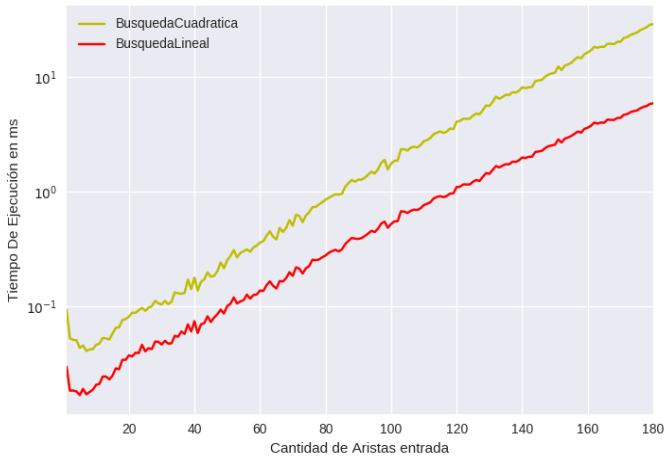
Este experimento nos sirvió para finalmente cerrar conceptos de cuál sería el límite entre la relación de lo que se le agrega a una heurística para "mejorarla". El considerar casos muy particulares en nuestra vecindad cuadrática lleva a que el poder de computo aumente significativamente, pero el resultado promedio en cambio no en mejora.

Experimento Complejidad Temporal Por Iteración

Para el primer experimento de esta sección queríamos mostrar la complejidad temporal de los algoritmos de Búsqueda Local dentro de cada iteración. Y como dijimos antes, esta sección la íbamos a hacer fijando cada variable para comprobar que el comportamiento teórico con respecto a cada una de ellas era correcto.

El primer gráfico corresponde al experimento realizado fijando la variable N, cantidad de nodos, en 20. Fuimos variando la cantidad de aristas dentro del grafo entre 0 y su capacidad máxima, que en este caso es de 180 aristas.

El resultado fue el siguiente:



Tiempos en Grafos random 20 Nodos y Aristas variables.

Como vemos el resultado fue el esperado, la función de crecimiento temporal claramente demuestra un gráfico lineal, al igual que estimamos dentro de la complejidad teórica para cada iteración.

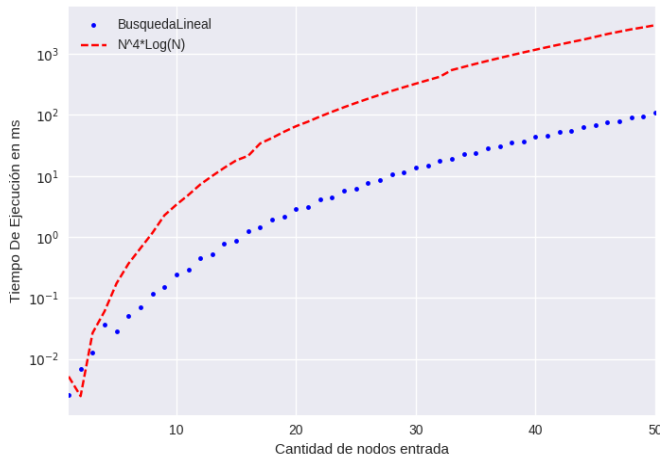
La segunda parte de este experimento la queríamos hacer fijando a M en un valor que nos permita mover la cantidad de nodos sin problemas. Sin embargo esto al primer intento no funcionó de manera exitosa.

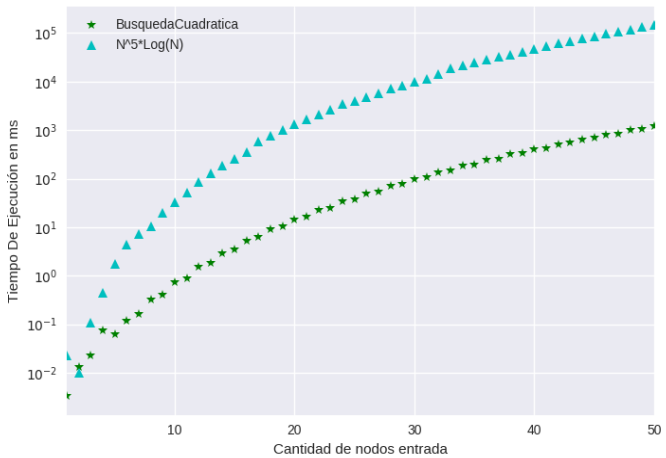
Nuestro algoritmo se basa en recorrer todos los vecinos y para cada uno de ellos aplicar una función en orden de n. Lo que resultaba que al aumentar la cantidad de nodos dentro del grafo disminuya la cantidad de aristas incidentes a los nodos de la solución ya que éstas se distribuyen entre varios otros nodos, lo que traía como consecuencia una reducción en el tiempo de complejidad de cada iteración significativamente.

Para resolver este problema optamos por tomar un M completo dentro de cada grafo. De esta manera sabemos que el tamaño de M es igual a un numero en relación a N, específicamente $\frac{n*(n-1)}{2}$. Así obtenemos que la complejidad teórica dentro de cada iteración pasa a ser de $O(n^5 * log(n))$ en caso de busqueda con vecindad Cuadrática, y $O(n^4 * log(n))$ en búsqueda con vecindad Lineal.

Ya definido esto pasamos a experimentar cuánto cuesta una iteración dado que se le pasa como $SolAct$ el grafo completo, que es una clique pues la cantidad de aristas es el máximo posible.

Los resultados fueron los siguientes:





Como era esperado, la pendiente de la curva en ambos casos es la misma, por lo que podemos concluir que el comportamiento temporal del algoritmo con respecto a la variable N es efectivamente el teórico.

5 Meta-Heurística GRASP

La meta-heurística que fue implementada para resolver el problema de hallar la clique con la frontera de aristas más grande posible, fue hecha con GRASP. GRASP (Greedy Randomized Adaptive Search Procedures) consiste en un algoritmo que construye soluciones por medio de un algoritmo goloso, con la excepción de que en cada iteración, en vez de deterministicamente hacer una elección de la mejor decisión a tomar, en base a algun criterio ya definido, GRASP crea una lista de candidatos a partir de los cuales elige aleatoreamente con cual de ellos avanzar en la construcción de su solución.

Este conjunto de candidatos se construye utilizando el mismo criterio que la heurística golosa. Analiza que vertices del grafo se pueden agregar a la clique actual, de tal manera que maximizen la frontera más que cualquier otro vertice.

Cuando la heurística llega a una potencial solución, luego intenta mejorarla aún más con el uso de una busqueda local.

Todo este procedimiento se repite una serie de veces que puede ser definida estatica o dinamicamente. Por ello nos referimos a lo siguiente:

- Corte estático: se le asigna una cantidad definida de iteraciones al ciclo GRASP.
- Corte dinámico: se le asigna una cantidad máxima de iteraciones del ciclo GRASP en las cuales esta permitido no devolver una mejor solución. Luego de k ciclos sin mejora en la solución, entonces la meta-heurística se detiene.

Para definir los candidatos a solución dentro de la construcción de la clique, se utiliza un parámetro alfa el cual es utilizado en la siguiente comparación para determinar que vertices son candidatos y cuales no. Consideremos la función f, que por cada $v \in V$ retorna en f(v) la cantidad de aristas que aportan a la frontera de la clique en el caso de unirsele. Si un vértice cumple la siguiente condición entonces ingresa en la lista de candidatos.

$$f(v) >= \max_{v_i \in V} f(v_i) - \alpha * [\max_{v_i \in V} f(v_i) - \min_{v_i \in V} f(v_i)]$$

5.1 PseudoCódigo

Este pseudocódigo corresponde a la implementación de la busqueda de la clique con mayor frontera, utilizando el corte dinámico para definir cuantos ciclos GRASP se correran antes de terminar la ejecución. La alternativa sería no resetear el contador de ciclos actuales cada vez que se encuentra una solución de mejor resultado.

Algorithm 11 CliqueMaxFrontGRASP

```
procedure CliqueMaxFrontGRASP(Graph grafo, int α , int iterGrasp )
    solActual ← clique vacía sin frontera
    ciclos ← 0
    while ciclos < iterGrasp do
        vecinos ← vector de longitud grafo.nodos
        CalcularGrados(vecinos)
        clique ← clique vacía sin frontera
        listaReducidaCandidatos ← vector vacio
        candidatos ← vector vacio
        grafo.AgregarSiHaceClique(clique, candidatos)      ▷ Agrega a candidatos si son adyacentes a toda la clique
        while candidatos.length > 0 do
            listaReducidaCandidatos.clear()
            max ← DevolveMaxCandidato(vecinos, candidatos)
            min ← DevolveMinCandidato(vecinos, candidatos)
            for cand ∈ candidatos do
                if EsCandidato(vecinos, cand, max, min, α) then
                    listaReducidaCandidatos.pushBack(cand)
            if listaReducidaCandidatos.length = 0 then
                Break;
            elElegido ← random(0,listaReducidaCandidatos.length)
            grafo.AgregarFrontera(clique, listaReducidaCandidatos[elElegido])      ▷ Actualiza la clique
            grafo.ContarVecinos(clique, vecinos) ▷ Agrega a vecinos cuantas aristas aportaría cada nodo a la frontera
            candidatos.clear()
            grafo.AgregarSiHaceClique(clique, candidatos)
        clique ← BusquedaLocalLineal(grafo,clique)
        ciclos ← +1
        if clique.frontera > solActual.frontera then
            solActual ← clique
            ciclos ← 0
    return solActual
```

5.2 Experimentación

Próximamente, en la Reentrega
solo el los labos del dc..